# DESCRIPCION DE UNA HERRAMIENTA DE PROGRAMACION PARA CELDAS DE MANUFACTURA

**Hernán Garzón Gutiérrez***
*hernangarzon@hotmail.com*

## 1. Introducción

Las redes de Petri constituyen una herramienta con un amplio fundamento teórico y gráfico, capaz de representar de manera adecuada y eficiente un sistema de tiempo real. Permiten modelar, simular y analizar el comportamiento de sistemas de manufactura que presentan procesos con paralelismo o concurrencia, sincronización y conflicto por recursos.

Según Desrochers [1], el modelamiento de sistemas manufactura por medio de redes de Petri está sustentado en los siguientes aspectos:

1. Las redes de Petri representan relaciones de precedencia e interacciones estructurales de eventos estocásticos, concurrentes y asincrónicos. Además, su naturaleza gráfica ayuda a visualizar la complejidad del sistema
2. Los conflictos y tamaños de los almacenes temporales (buffers) pueden ser modelados fácil y eficientemente
3. Los bloqueos en el sistema pueden ser detectados
4. Las redes de Petri representan una herramienta de modelamiento jerárquico con una fundamentación matemática y práctica bastante desarrollada
5. Los modelos ofrecen una estructura organizada para llevar a cabo un análisis sistemático de sistemas complejos

* Matemático, Magister en Ingeniería de Sistemas Universidad Nacional de Colombia. Profesor adscrito a la Facultad Tecnológica de la Universidad Distrital F.J.C.

6. Finalmente, los modelos de redes de Petri pueden ser utilizados en la implementación de sistemas de control en tiempo real para un sistema de manufactura flexible FMS. En consecuencia, reemplazan a los controladores lógico-programables.

Este artículo hace énfasis en el modelamiento de la Celda Didáctica de Manufactura Flexible (CDMF), un prototipo construido con fichas de Lego y Fishertecnick. El esquema de la celda construida en EIDOS[1], con propósito didáctico, se muestra en la Figura 1; consta de un robot, un carro, un torno, un taladro, una fresadora, una estación de carga y otra de descarga, representados por elementos como un plotter y una mesa rotatoria con cuatro posiciones.

Las operaciones consideradas en la celda tienen unos tiempos determinados de manera exacta; por esto las redes de Petri de tiempo determinístico son el modelo apropiado en este caso.

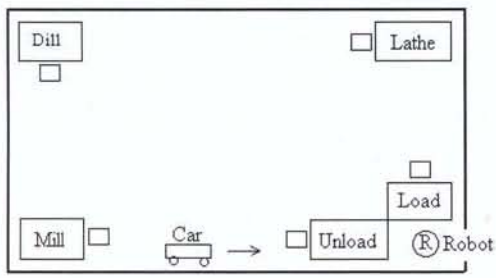



**Figura 1: Celda Didáctica de Manufactura Flexible**

## 2. Redes de Petri y Modelamiento en la Celda

### 2.1 Definiciones

Desrochers [1] presenta la siguiente definición:

Red de Petri. Una red de Petri ordinaria es un grafo dirigido representado por una cuádrupla de la forma $N = \{P, T, I, O\}$, donde:

$P=\{p_1,p_2,...p_n\}$ es un conjunto finito de lugares
$T = \{t_1, t_2,...,t_n\}$ es un conjunto finito de transiciones.

$I$ es una aplicación de entrada $T \times P \rightarrow \{0,1\}$ correspondiente a un conjunto de arcos dirigidos de T a P; es una aplicación de salida $T \times P \rightarrow \{0,1\}$ correspondiente a un conjunto de arcos dirigidos de T a P.

Las redes de Petri forman una estructura dinámica que está dada por la marcación y su evolución (regla de disparo). Así, en forma adicional a lo descrito anteriormente, se le pueden asociar fichas (tokens) en cada lugar, a través de una aplicación $M_0$, denominada marcación inicial. Si p representa un lugar, el número $m_0$ (p) se interpreta como el número inicial de fichas en ese lugar.

**Marcación.** En Dicesare [2] se define una red de Petri marcada como una dupla $MP = \{R, M_0\}$ donde:

$R$ es una red de Petri ordinaria
$M_o: P \rightarrow N$ es una aplicación que asocia a cada lugar un número de fichas
La aplicación $M_0$ se denomina marcación de la red.

**Regla de disparo.** La marcación en una red evoluciona de acuerdo a un juego de fichas (token game) que se rige por la siguiente regla de disparo:

1 Grupo de Investigación del Departamento de Sistemas de la Facultad de Ingeniería de la Universidad Nacional de Colombia

1. Una transición se dice habilitada para una marcación, si cada lugar de entrada tiene al menos una ficha. Es decir:

$$(\forall pi \in Inp(t)(m(pi) \geq 1)$$

2. El disparo de una transición habilitada "t" es una operación instantánea que cambia la marcación, de tal forma que disminuye en una unidad el número de fichas en cada lugar de Inp (t1) y aumenta en una unidad el número de fichas en cada lugar de $Out(t)$

3. Ante la eventualidad de dos transiciones, $t_1$ y $t_2$ habilitadas en forma simultánea, es decir, cuando existe un lugar " p", que pertenece a los conjuntos Inp (t1) y Inp(t2) y al mismo tiempo, un mecanismo de reso lución de conflictos dirime la situación.

**Redes de Petri de tiempo determinístico** [3]. Las redes de Petri temporales (Timed Petri Nets TPNs) se introducen a través de retardos asociados con las transiciones que están dados por un intervalo de tiempo finito o son transiciones inmediatas. Así, tenemos que:

Una red de Petri de tiempo determinístico es una dupla de la forma $DPN = \{R, D\}$ donde
$R$ es una red de Petri ordinaria
$D : T \rightarrow N$ es una aplicación que asocia a cada transición un tiempo de disparo.

## 2.2. Modelamiento

La Figura 2 muestra un modelo de red de Petri para un trabajo sobre la celda, donde diez piezas de materia prima deben ser cargadas y procesadas en paralelo (concurrentemente). A cinco de ellas se les hace una operación en el torno y a las otras cinco una operación en el taladro antes de abandonar el sistema.

Los lugares (círculos de borde negro) representan operaciones o recursos disponibles. Las transiciones (barras horizontales) representan el evento que comprende el inicio y el final de una operación. Los números dentro de los círculos indican la cantidad de fichas, es decir la marcación inicial.

La semántica asociada al modelo está dada por:

Lugares:
**Input.** Disponibilidad de piezas sin procesar
**Robot.** Disponibilidad del robot
**Load.** El robot carga la pieza en la estación de entrada
**Car.** Disponibilidad del carro
**Lathe.** Disponibilidad del torno
**Drill.** Disponibilidad del taladro
**Transport.** Los lugares demarcados con (tp) corresponden a operaciones de transporte hacia las estaciones de maquinado o hacia la estación (unload) de descarga
**lathe Operation.** El torno procesa la pieza
**drill Operation.** El taladro procesa la pieza
**Unload**. El robot descarga la pieza procesada y la coloca en la estación de salida

Dos lugares adicionales, con cinco fichas cada uno, aparecen en la parte superior de la Figura. Ellos hacen que sólo cinco piezas de cada tipo sean procesadas.

**Transiciones:**

Los eventos que disparan las transiciones, así como sus respectivos tiempos de disparo, son los siguientes:

**$t_1$:** Comienzo de carga de la pieza en la estación de entrada (4 seg.)
**$t_2$:** Comienzo de transporte al torno (2 seg.)
**$t_3$:** Comienzo del proceso en el torno (8 seg.)

$t_4$: Comienzo de transporte a la estación de salida (2 seg.)
$t_5$: Comienza descarga de la pieza procesada (4 seg.)
$t_6$: Comienzo de transporte al taladro (2 seg.)
$t_7$: Comienzo del proceso en el taladro (10 seg.)
$t_8$: Comienzo de transporte a la estación de salida (2 seg.)
$t_9$: Comienza descarga de la pieza procesada (4 seg.).

Para la construcción de la red que se muestra en la Figura 2 se utilizaron las técnicas de síntesis descritas en [4]. El modelo fue editado y ejecutado con la herramienta petriProgrammer.

## 3. PetriProgrammer

El software PetriProgrammer es una herramienta de programación desarrollada para simular la operación del prototipo CDMF. PetriProgrammer está basado en una red de Petri de tiempo determinístico, que facilita expresar al usuario las especificaciones de un problema de producción dado.
La aplicación, construída en lenguaje Modula-3, corre bajo Linux. Para implementar la interfaz de usuario GUI se utilizó la librería Trestle; esta es un conjunto de interfaces que giran alrededor de un tipo de dato abstracto VBT (virtual bitmap terminal) [5][6][7] que permite el acceso al sistema X Windows. Los modos en que petriProgrammer opera son: edición y programación de celda.

### 3.1. Edición

El usuario puede, a través de una de las ventanas del programa, entrar y editar las especificaciones de una red de Petri. Por medio de una barra de herramientas puede generar lugares, transiciones y arcos dirigidos. Adicional-mente, pueden ser agregadas características como el número de fichas de un lugar (es decir la marcación inicial) y el tiempo en segundos de una transición.

### 3.2. Programación de Celda

Por medio de otra ventana de la aplicación, operando en el modo programación de celda, el usuario puede especificar la secuencia de operaciones a través de una barra de herramientas en la parte superior de esta ventana. El programa dado por el usuario está formado por esta secuencia en la que, además de las estaciones

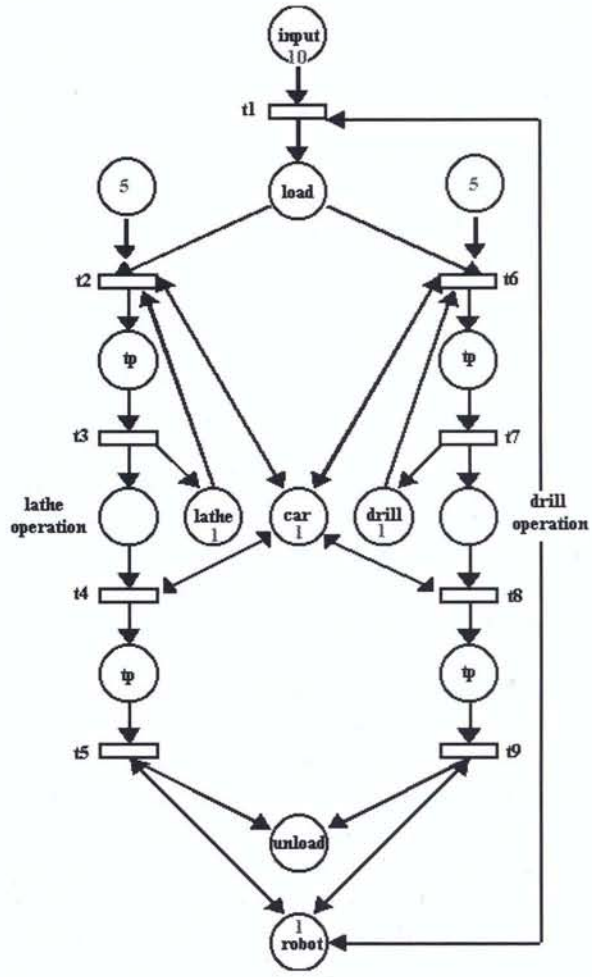



**Figura 2. Procesamiento de Piezas en Paralelo**

de trabajo *lathe, drill y mill*, se determina el tipo de programa que debe correrse en cada una de ellas.

Estos programas están determinados por las cantidades de tiempos que consume cada operación, que pueden ser modificada por el usuario, el cual debe también determinar la cantidad de piezas que se van a maquinar en la celda. Mediante un algoritmo, el programa tiene la capacidad de generar de manera automática una red de Petri que modela el problema de producción introducido por él.

Este modo tiene algunas limitaciones. Por ejemplo, se pueden programar tareas en las que sólo se procesa un único tipo de pieza, esto es, cuando diversas unidades están sometidas a una misma secuencia de operaciones.

### 3.3. Simulación

El programa puede ejecutar una red de Petri una vez ésta es introducida por medio del editor o generada automáticamente por el programador de celda. Al ejecutar la red se inicia un flujo de fichas a través de ella a partir de una marcación inicial que representa el desarrollo de las operaciones, el procesamiento y transporte de las piezas en la celda. El usuario puede observar en una ventana la simulación que ejecuta la red y en otra una serie de estadísticas, que constituyen indicadores del desempeño del sistema.

## 4. Algoritmo de Ejecución

Una red de Petri es esencialmente una estructura matemática que representa un sistema en el cual una serie de precondiciones, esencialmente locales, deben cumplirse para alcanzar un estado. La secuencia de estados, ocasionada por el disparo de las transiciones de manera concurrente, determina la ejecución de la red. Para que un sistema de cómputo pueda simular su ejecución debe captar la naturaleza concurrente del problema. El lenguaje Modula-3 posee esta característica; a través de hilos de programación (threads) [8] se pueden obtener las propiedades, que debe tener el algoritmo de ejecución.

### 4.1. Hilos de Programación (threads) en el Problema de los Filósofos

Un problema clásico de programación concurrente, denominado "el problema de los filósofos", es utilizado por Boszormenyi [9] para ilustrar la utilidad de los hilos (threads) en modula-3. Este problema plantea lo siguiente: cinco filósofos se sientan en una mesa redonda, según el esquema de la Figura 3. Cada uno tiene

**Figura 3: Problema de los filósofos**

un plato de arroz, que come utilizando un par de palillos; en medio de dos platos hay un palillo. La vida de un filósofo consta de períodos alternados de comer y pensar. Cuando siente hambre come algunos bocados siempre y cuando estén disponibles los palillos izquierdo y derecho, luego piensa por algún tiempo hasta que nuevamente siente hambre.

En la solución planteada para este problema cada filósofo es representado por un thread (hilo). Cuando un hilo es activado este llama al procedimiento *Start,* que a su vez llama al procedimiento *Philosopher,* el cual comienza un ciclo (*loop*) que consta de dos procedimientos, *eat* y *think*, que permiten el cambio en una variable de estado para cada hilo; estos procedimientos simulan las acciones de comer y pensar, haciendo una pausa de duración aleatoria.

Mientras un hilo está ejecutando un procedimiento con una pausa, los demás hilos siguen corriendo en forma concurrente.

### 4.2. Threads en el Algoritmo de Ejecución

El problema de la ejecución de la red de Petri es similar al problema de los filósofos. Este come siempre y cuando se cumplan unas precondiciones: que no esté pensando y que los palillos derecho e izquierdo estén disponibles. En el caso de una transición (ver Figura 4), ésta es disparada sólo cuando se cumplan dos precondiciones: que no éste siendo disparada en ese instante y que en cada lugar de entrada exista por lo menos una ficha. En ambos casos se simulan los estados mediante una pausa. La solución dada al problema de los filósofos fue utilizada para construir el algoritmo de ejecución de la red de Petri para la aplicación.

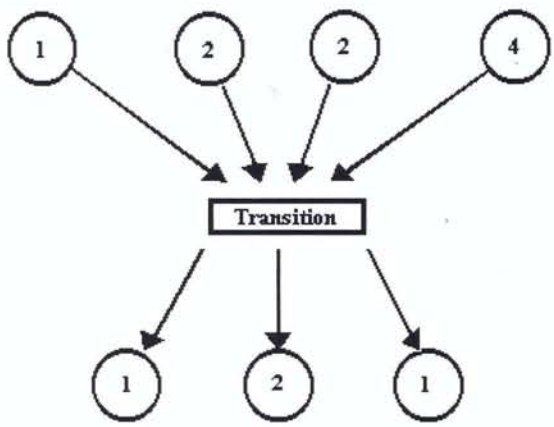



**Figura4: Transición en una Red de Petri**

Un módulo denominado *Algorith* en *PetriProgrammer* se constituye en el ejecutor de la red de Petri. Este define closure como un subtipo de *Thread* en el cual el método *apply* es reescrito en el bloque *overrides* de su declaración, como puede observarse al principio del código del siguiente programa:

```
MODULE Algorithm;
IMPORT Thread, Tick , SIO, Text, Word,
IntList;
IMPORT petriExecutor, PetriNet, Vertex;
FROM Colors IMPORT black, red;
TYPE
  Closure = Thread.Closure OBJECT
   id: CARDINAL;
   state := States.waits;
  OVERRIDES
   apply := Start
  END;
States = {waits, fires, finish};
BEGIN
END Algorithm.
```

La enumeración *States* aquí definida permite asignar los estados de cada hilo. Los hilos son generados en el procedimiento *MakeThreads* sobre la lista de transiciones del grafo que representa la red, a través de la sentencia *EVAL Thread.Fork(cls[idThread])*.

```
PROCEDURE MakeThreads() =
  VAR
    idThread : INTEGER;
  BEGIN
    FOR i:=0 TO threads-1 DO
                        id_thread     :=
IntList.Nth(transitionList,i);
      cls[id_thread] := NEW(Closure, id :=
idThread);
       EVAL Thread.Fork(cls[idThread]);
    END;
  END MakeThreads;
```

El método *apply* es personalizado a través del procedimiento *Start,* el cual llama al procedimiento *transition*, que define un programa secuencial que se ejecuta en cada hilo.

```
PROCEDURE Start(self: Closure): REFANY
=
 BEGIN
   Transition(self.id); RETURN NIL
 END Start;
PROCEDURE Transition(id: INTEGER) =
 BEGIN
  LOOP
    IF NOT finish THEN
     PickUp(id);
     Fire(id);
     PutDown(id);
     ELSE NewState(id, States.finish); EXIT
    END;
  END
 END Transition;
```

Tres procedimientos forman el ciclo que determina cada hilo.

El procedimiento *Pickup* verifica que la transición esté habilitada, esto es, si *avail(id)* es diferente de cero, en cuyo caso se inicia su disparo por medio del procedimiento *UpdateInputPlace*. Este procedimiento disminuye en una unidad las fichas en cada lugar de *inp(id)*, esto es, del conjunto de lugares de entrada de la transición. En caso contrario el semáforo *mutex* detiene el procedimiento hasta que la condición available se satisfaga.

```
PROCEDURE PickUp(id: INTEGER) =
  BEGIN
    LOCK mutex DO
          WHILE avail[id] < 1 DO
Thread.Wait(mutex, available) END;
     INC(workInProcessing);
     UpdateInputPlace(id);
    END;
  END PickUp;
```

El procedimiento *Fire* cambia el estado de la transición a *fires*, y hace una pausa que dura el tiempo asociado a la transición.

```
PROCEDURE Fire(id: INTEGER) =
  VAR
    vertex : Vertex.T; idVertex :INTEGER;
  BEGIN
    NewState(id, States.fires);
    idVertex := (sizeGraph-1)-id;
   vertex := PetriNet.Nth(graph, idVertex);
  Thread.Pause(FLOAT(vertex.data, LON-
GREAL));
  END Fire;
```

El procedimiento *PutDown* continúa el disparo de la transición, actualizando las fichas de todos los elementos de *outp(id)*, esto es, del conjunto de lugares de salida de la transición, con el procedimiento *UpdateOutputPlace*, que aumenta en una unidad las fichas en cada lugar de *outp(id)*.

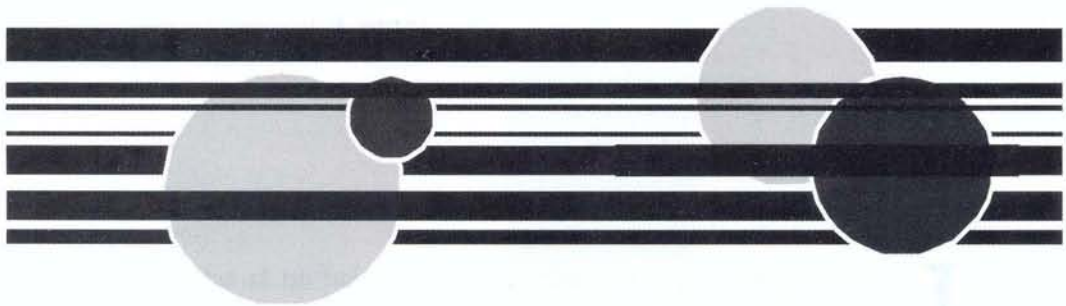
```
PROCEDURE PutDown(id: INTEGER)=
   BEGIN
     LOCK mutex DO
       DEC(workInProcessing);
       UpdateOutputPlace(id);
        Thread.Broadcast(available);
      Finish();
    END;
   END PutDown;
```

Finalmente este procedimiento concluye el ciclo terminando el disparo de la transición y colocando el hilo en el estado *waits*, esperando nuevamente a ser disparado.

## 5. Conclusiones

En este trabajo se analizó el modelamiento de la celda (CDMF) en un escenario determinístico y en el plano de la simulación. En estudios posteriores debe abordarse el problema del control de la celda, para lo cual este trabajo despeja una buena parte del camino.

La generación automática de la red de Petri es uno de los mayores aportes de este trabajo. Queda abierta la posibilidad de involucrar agentes inteligentes que permitan modelar sistemas complejos, posiblemente con tiempos de disparo distribuidos aleatoriamente que incluya eventos como las fallas en las máquinas.



## REFERENCIAS BIBLIOGRÁFICAS


- DESROCHERS, Alan A. y AL-JAAR, Robert, Applications of Petri Nets in Manufacturing Systems, NewYork : IEEE Press, 1995.
- DICESARE, Frank. Practice of Petri Nets in Manufacturing. London, England: Chapman & Hall, 1993.
- BEHERA, T.K. et al. Modelling and Performance Evaluation of Flexible Manufacturing Systems Using Deterministic and Stochastic Timed Petri Nets. Indian Istitute of Science, Bangalore, India. 1995.
- ZHOU, Meng-Chu, DICESARE, Frank Petri Net Synthesis for Discrete Event Control of Flexible Manufacturing System , Boston, MA: Kluwer Academic, 1993.
- MANASSE Mark and NELSON Greg, Trestle Reference Manual, SRC Report 68, DEC Systems Research Center, Palo Alto, california, December 1991.
- MANASSE Mark and NELSON Greg, Trestle Tutorial, SRC Report 69, DEC Systems Research Center, Palo Alto, california, May 1992
- STANSIFER Ryan, Trestle By Example, October 1994. en http:// www.research.digital.com.
- BIRRELL, Andrew, An Introduction to Programming with Threads, SRC Report 35, DEC Systems Research Center, Palo Alto, california, January.1989.
- BOSZORMENYI Laszlo, WEICH Carsten, Programming in Modula-3, Springer. Verlag, 1996.